Speedy Stores: Benchmarking the performance of key/value databases in C-based embedded systems.

Camden Fergen
Iowa State University
Ames, IA
cfergen@iastate.edu

Simon Weydert
Iowa State University
Ames, IA
weyderts@iastate.edu

Abstract

We analyze how C-based key/value databases perform on a resource-constrained embedded system. The restricted system used in this paper is the BeagleBone Black. We collect and analyze the performance of BerkeleyDB, Lightning Memory-Mapped Database (LMDB), and UnQLite in terms of processing speed, latency, throughput, and storage space. We find that all database files remain the same size for a set number of keys. We also find that the performance on UnQLite and BerkeleyDB is very similar, whereas LMDB excels with read but falters with write operations.

1 Introduction

Embedded systems are becoming essential to modern technology, powering everything from smart home devices and smartphones to automotive and industrial devices. These systems are typically constrained by their limited processing power, memory, and energy requirements. Often, these devices operate in real-time environments, meaning that fast and efficient data access is critical to functionality.

The number of embedded devices continues to grow, with an estimate projecting the number of devices to exceed 27 billion by 2025 [11]. These systems must store and analyze their data locally for various reasons such as unreliable Internet connectivity or strict privacy requirements [4], [8]. This rapid growth has increased the demand for lightweight and efficient data storage solutions within embedded operating systems.

Although embedded systems account for more than 98% of all computing devices worldwide [9], there remains a lack of comprehensive performance comparisons between different databases designed for embedded systems. While individual studies are available on specific databases or optimization techniques [1]–[3], [10], few have comprehensively evaluated multiple databases under the same conditions.

In this paper, we benchmark three popular C-based embedded databases: BerkeleyDB, LMDB, and UnQLite. We use the BeagleBone Black as our embedded system to mimic a real resource-constrained device. We use the Yahoo! Cloud Serving Benchmark (YCSB) framework, with custom database interfaces, to evaluate read/write throughput, latency, and storage behavior under various workloads. Our goal is to be able to offer a practical comparison for developers looking to select a database for embedded applications.

The rest of the paper is organized as follows. First, we provide a background on embedded system databases and why performance matters on these systems. Then, we provide our motivations for studying embedded systems databases, using the YCSB benchmarking framework, and our choice to use the Beagleboard Black. After that, we describe our methodology for developing, configuring, and deploying benchmarks for our target databases on the Beagleboard. Finally, we present our findings about which databases performed the best and related observations, followed by a discussion of potential further work.

2 Background and Motivation

In this section, we describe our motivations for studying and evaluating embedded databases within constrained environments and the relevance of embedded systems within modern computing.

2.1 Embedded databases

Embedded databases differ from traditional client-server databases in that they are entirely self-contained and designed to operate within the same address space as the host application. Unlike their server-based counterparts, which require a constant network connection, embedded databases offer a lightweight, low-overhead solution, ideal for limited resource systems.

These databases are well-suited for embedded environments as they result in a low processor overhead and provide quick data access. In situations where these systems must operate without any reliable internet, such as remote environments, they remove any need for a networked database. Additionally, due to their simplicity and speed, they make a great fit for real-time applications where quick data access and minimal latency is crucial.

2.2 Why Embedded Systems Matter

Embedded systems are a form of dedicated computing platforms designed to perform specific tasks. They usually contain constrained hardware, such as low-power CPUs, small memory footprints, and tight energy budgets, which dictates how to design software for them, especially in terms of data access and storage.

A major driver of embedded systems today is the new realm of Internet of Things (IoT) devices. These devices typically generate large volumes of data, whether that be in sensor collection or pictures taken, which must be processed and stored locally typically due to bandwidth or privacy concerns. Edge computing has also driven this further by pushing data processing closer towards the source to reduce the latency and cost of cloud computation [5], [6].

2.3 Motivation for This Study

Although many embedded databases have been analyzed individually, few studies perform comparisons under the same uniform testing conditions. Developers building new embedded applications often face uncertainty when deciding on which database to use due to the lack of solid benchmarking data.

Our study hopes to address this gap by comparing multiple embedded databases using the same hardware, datasets, and testing framework to ensure a fair evaluation of performance metrics such as: latency, read/write throughput, and storage efficiency. We hope to offer a solid insight that can help guide embedded developers to determine the best database to suit their needs.

2.4 Database Selection

We want to choose lightweight databases, as heavier databases may consume too many resources on a constrained device. We chose key/value stores because they are incredibly simple with minimal storage and performance overhead. Additionally, the simplicity of key/value storage makes comparing databases easier, since they all share the same set of key/value operations, such as reading from keys and inserting/changing key/value pairs. We chose databases that are lightweight and operate primarily as key/value stores. The three we chose to investigate are BerkeleyDB, LMDB, and UnQLite.

There are some databases we decided to forgo. Relational databases like SQLite can act as key/value stores, but that is not its primary function, and as such run the risk of degraded performance in comparison to simpler databases with fewer features. Object-based stores such as ObjectBox run the same risk of reduced performance because of their more advanced features.

2.5 Benchmarking Selection

In choosing a benchmarking tool, we needed a tool that was built to add new databases. All of the databases we are testing don't have comprehensive benchmarking support; they aren't the powerful cloud- or server-oriented databases that are more commonly benchmarked. Our tool needs to support key/value database operations and be small enough to fit on the target machine.

We chose YCSB as our benchmarking tool. While it was initially built to support cloud-based relational databases, its approach to database interfaces is versatile enough to be re-framed as key-value operations for local databases (this process is detailed in the Methodology section). YCSB can be built and compressed into JAR files and then executed on any other platform without having to worry about cross-compilation conflicts, as long as the target can run Java.

Another promising candidate for our benchmarking tool was BenchDB, but it does not have as much support for adding databases as YCSB does, nor is it as flexible as YCSB.

3 Methodology

We describe our methodology to benchmark databases on the BeagleBone Black. For each database, we write wrapper code for the database so that it can interact with YCSB. Then we side-load the database source code to the embedded system to compile it on that system. Finally, we copy the JAR-executable YCSB testing framework and supporting scripts to the BeagleBone, then run tests. For simplicity, we are only testing the performance of these databases as key/value stores.

3.1 Hardware Setup

To replicate a realistic embedded system environment, we chose to run our tests on the BeagleBone Black development board. This board features a 1GHz ARM Cortex-A8 processor, which are commonly used in embedded systems [14]. It also includes 512MB of DDR3 RAM, providing a constrained memory environment, and 16GB of storage via a SanDisk SD card. Together, these specifications help to emulate a the specifications of a typical embedded system.

For our operation system, we opted to use the provided Debian Linux image that BeagleBoard provides on their website. While Debian is not usually considered a pure embedded operating system due to its larger overhead, it can be often found on development boards like the BeagleBoard Black due to its extensive hardware support and familiar development environment. This made it a practical choice for our benchmarking purposes, even while more lightweight or real-time operating systems are typically found in production.

3.2 YCSB Compatibility

YCSB is a large Maven Java project. It consists of a top-level project directory that contains many second-level project directories, or modules. Each database that can interface with YCSB has a database binding written as a Maven proeject placed in one of these modules. The core YCSB functionality is also a project module. To add a new database, the following must be achieved.

- Add a module as a new maven project directory placed in the root project directory.
- Add the new module's name to the root POM.XML.
- Write a YCSB database binding class unique to your database.
- Build the module as a Maven project with its YCSB dependencies.

Each database binding must have a client class that extends the YCSB DB class. This class provides the fundamental functionality that YCSB requires to function. For our purposes, the DB class must implement the following methods:

- **init():** Used to initialize the database used for testing. In our environment, it creates a new database file if it doesn't exist or open an existing database file.
- **cleanup():** Used to close the database before terminating a YCSB test.
- **insert():** Used to insert a key/value pair into the database.
- **update():** Used to update a particular set of fields for a record. Since each of our records has only one value, this is identical to the insert method.
- **read():** Used to fetch a value from a key. In our case, the value can be any arbitrary length binary value.
- **delete():** Used to remove a key/value pair.
- **scan():** Used to look for specific fields within a record. Since each of our records has only one value, this method goes unimplemented.

Our databases are C-based and run inside of applications rather than as services. Since YCSB typically benchmarks cloud databases that run as services with dedicated request and authentication interfaces, we must write an additional class to bind C library functions to a set of java methods that can be called by the YCSB client. To achieve this, we've elected to compile the the C databases into a Shared Object file (a .so file, the Linux equivalent of a .dll Windows file), which can be fetched at runtime by the Java Native Access (JNA) library. A JNA library class is written for each database, and each must implement the following functionality.

- Open the database.
- Close the database.
- Store a key/value pair.
- Fetch a key/value pair.
- Delete a key/value pair.

BerkeleyDB, LMDB and UnQLite each have their own unique API quirks. For example, whereas UnQLite provides simple function calls that are easy to bind JNA to, BerkeleyDB instead provides its core functions as function pointers accessible from a database struct. Since JNA does not handle this kind of access well, a helper C file is created for each database as its C API. This helper function implements the same five functionalities listed above. Writing the C helper API has the benefit of simplifying the JNA libraries and making each database's JNA library implementation (and as a result its YCSB client implementation) nearly identical.

3.3 Compilation Procedures

For each database, we build a minimal testing framework. This is a small directory that can be copied to BeagleBone in favor of copying the entire YCSB source code. We do this because copying the large YCSB source code to the BeagleBone takes a long time, and because most of the modules listed in the source code are for databases we aren't testing. The minimal testing framework will contain the YCSB workloads, the shell scripts used to load and execute YCSB workloads, and the JAR files for the database binding and its dependencies.

Before compacting the project into JAR files, the database module and its dependencies are built with Maven and use the YCSB CommandLine program to test that opening the database, inserting, reading and deleting key/value pairs, and closing the database all function as expected. After the database binding is proven to function correctly, the database module and its dependencies are packaged into JARS, and those jars are copied into the minimal testing framework directory.

The rest of the minimal testing framework is populated by workloads and helper shell scripts. Two of the workloads, Workload A and Workload C, are taken from the small set of YCSB default workloads. Workload A is 50/50 read and update, while Workload C is entirely read. The last workload, Workload U, is purely updates, providing a counterpoint to Workload C to compare the speeds of updates and reads. The helper shell scripts run Java, call YCSB on each on each database client, and either load or run a workload. In order for Java to run on the BeagleBone, we use a JDK compatible with ARM version 7 hard-float machines (such as the Beaglebord) publicly available from the Azul Zulu website [13].

3.4 Running Tests on the BeagleBone

To avoid cross-compilation errors, the .so files required for each database's JNA interface are generated on the BeagleBone. The necessary source files (and make files in the base of BerkeleyDB) are temporarily copied to the BeagleBone where the .so files are generated and placed in a location hard-coded into the JNA interface. For our purposes, we chose the new directory /tmp/shrLib. After a .so file is generated, the source code is discarded.

Since the testing framework is written entirely in Java, there is no need to build the Maven project locally on the BeagleBone. We instead copy the minimal testing framework directory, which is around 6MB for both databases, to the BeagleBone.

Once the board has the .so file and the testing framework, it's time to run tests. To start testing a workload, we delete the previous workload's database file if it exists. For each test, we run the workload 10 times, taking measurements of performance and the size of the database file after each run. Each workload run consists of 1000 records containing a single 100 byte field each. For each test, 1000 operations are performed on the database. The operations themselves are randomly generated based on the YCSB workload. This simulates a small burst of operations on a small key/value database, and allows us to measure how the size of a database changes over time.

4 Results

In this section, we compare the performance of three databases: BerkeleyDB, LMDB, and UnQLite. For each test, we average the overall runtime, overall throughput, read latency, and write latency metrics over all 10 workload runs. The latency averages are not perfect for workload A, since their distribution is randomized, but always very close to a

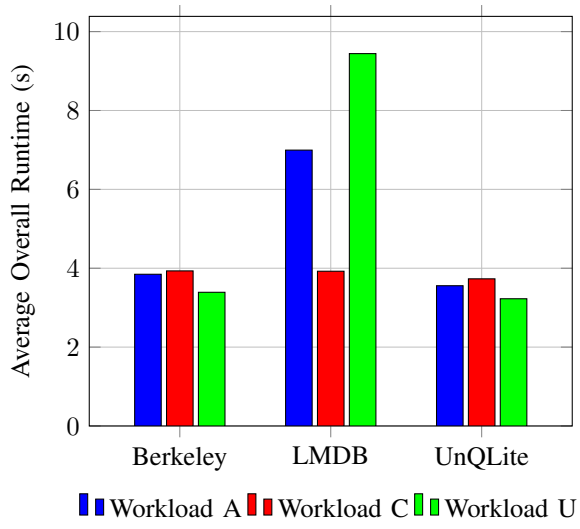

Fig. 1: Overall Runtime Comparison Across Databases and Workloads

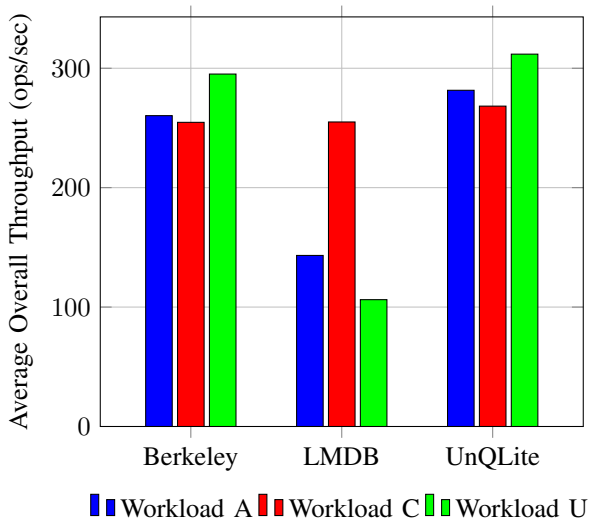

Fig. 2: Overall Throughput Comparison Across Databases and Workloads

perfect 50/50. We also measure the size of a database's file after each workload run.

4.1 Runtime and Throughput

The average overall runtime and throughput for each test are indicated in Figure 1 and Figure 2, respectively. The throughput is calculated as the inverse of runtime, so each is really just a different representation of the same metric. As YCSB calculates, the overall runtime is the total runtime of the YCSB test, not the aggregated runtime of each database function call. Thus, there will be some overhead from the YCSB tool in the calculation of runtime and throughput.

For all workloads, BerkeleyDB and UnQLite have similar performance. The performance of Workload C on LMDB is comparable to the performance of the BerkeleyDB and UnQLite workloads. The performance of Workload A and Workload U on LMDB are twice to three times worse than the other databases' workloads. This is to be expected, since LMDB is optimized for read-performance.

4.2 Latency

Latencies are measured between the initial request to the database and its response to the worker thread submitting the database requests. Since there is only one worker thread, context switching is a minimal or nonexistent issue, and YCSB overhead is much more negligible since the thread records each operation's latency.

The read latency is indicated for workloads A and C in Figure 3. Workload U is an update-only workload, so it does not have a read latency. Similarly, the update latency is indicated for workloads A and U in Figure 4. Workload C is a read-only workload, so it does not have a read latency.

The read latency of BerkeleyDB was generally worse than the read latency of LMDB and UnQLite, which were comparable. LMDB had the best read latency for Workload A, whereas

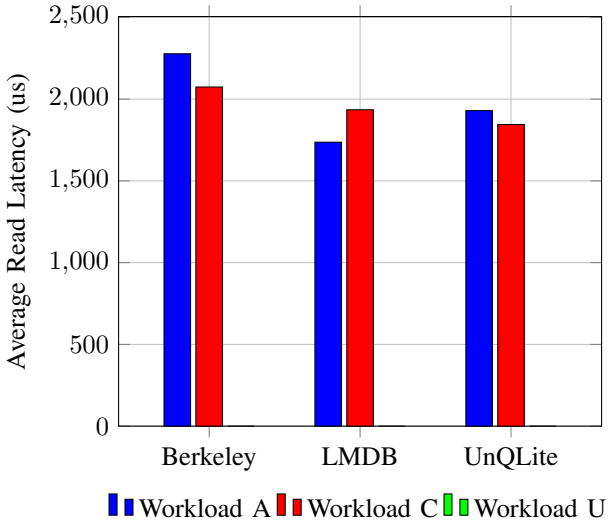

Fig. 3: Read Latency Comparison Across Databases and Workloads

UnQLite had the best read latency for Workload C. This is somewhat surprising, because LMDB is built explicitly for frequent reads, and Workload C is a pure read-only workload.

The write latency of LMDB is far worse than the that of BerkeleyDB and UnQLite. The write latencies of BerkeleyDB and UnQLite are very similar. For all databases, the performance of Workload U exceeded that of Workload A.

4.3 Disk Usage

The disk usage of all three databases throughout the test are plotted in Figure 5. Each size of all three database files did not grow over the duration of the test. The workload chosen did not affect the size of the database for any of the databases. The Berkeley database file took up the least space, and the size of

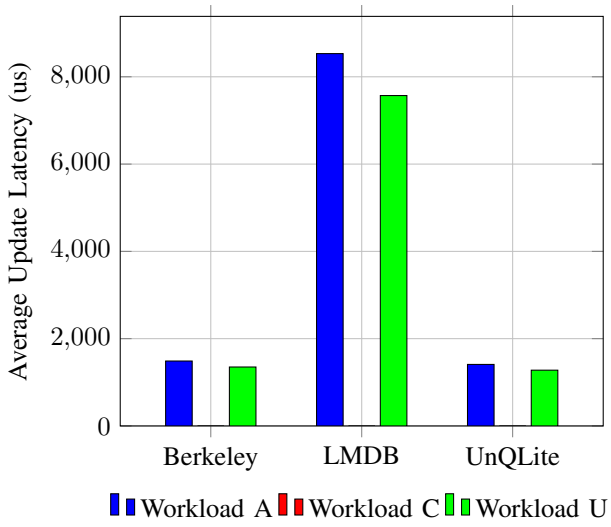

Fig. 4: Update Latency Comparison Across Databases and Workloads

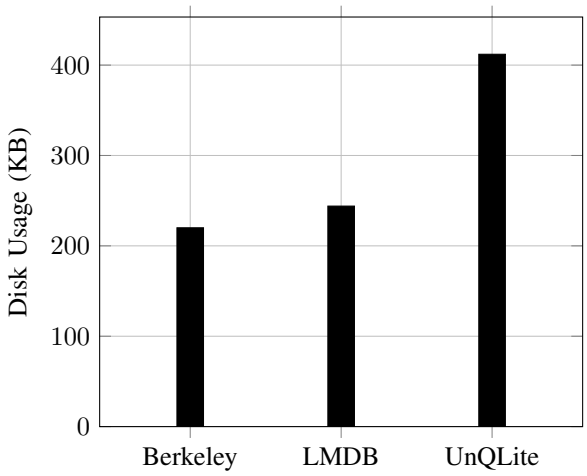

Fig. 5: Disk Usage Comparison Across Databases

the LMDB database was not too much larger. UnQLite created a significantly larger file.

5 Conclusion

We can make observations about BerkeleyDB, LMDB, and UnQLite only in this specific environment. Each database runs on a Beaglebone Black, a restricted system. The databases were operated on with a randomized keyspace of 1000 records and a total operation count of 10000 operations. In this environment, BerkeleyDB and UnQLite showed relatively similar performance with good behavior in both read and write operations.

5.1 Further Work

To best study the effects of certain databases on realistic application loads, we should do an assessment of real database workload behaviors on several embedded systems in real-world applications. That way, we can construct database operation proportions that more closely match real world applications. We could also play around with the distribution of operations made on keys. YCSB supports several distributions based on how recently a database element was accessed, such as targeting keys with recent changes far more than others.

We only used one YCSB worker thread because the target system only has one processor core. Moreover, for constrained systems, there are typically only a few programs, or perhaps only one primary program, running at a time, which would make multi threading unusual. For other systems, it may be beneficial to test the effects of running multiple YCSB worker threads to simulate several threads or processes querying the same database.

We tested a relatively small number of operations and a small key-space on each database. It would be beneficial to see how the difference in performance between each database changes as the key-space and number of operations increases. For instance, LMDB may see performance increases over longer tests because it maps the database file to a process address space, whereas BerkeleyDB and UnQLite simply make file system calls.

A cross-compilation method would make the deployment of database bindings and library files to a target device much faster. This would facilitate adding new databases with larger source codes, since the library file compiled form a database is much smaller than the source code it is compiled from.

It would be wise to find or develop a benchmarking tool that runs outside of the environment of an embedded system and monitored it over a network connection or a physical connection. While this may introduce network overhead, as long as the embedded system could be connected to the benchmarking tool, the embedded system would behave more similarly to a genuine device with a real application. This would necessitate some sort of light API layer for each database function for the embedded system that could accept remote requests. This would work well for a our tests because each database incurs the same network delay, since they run on the same machine connected directly to the benchmarking client.

We constrained our focus to key/value stores, but there are other kinds of database storage methods out there. There is object-based storage, which stores data in a programmer-defined schema. This might benefit programmer who need to store specific data structures and guarantee their atomicity. There are also relational databases like SQLite that would be prime candidates for testing given their popularity and prevalence in many applications and which support writing and reading to the individual fields of a record.

References

[1] Ouarnoughi, H., Boukhobza, J., Olivier, P. *et al.*, "Performance analysis and modeling of SQLite embedded databases on flash file systems." *Design Automation for Embedded Systems*, **17**, 507–542 (2013). https://doi.org/10.1007/s10617-014-9149-2

[2] Malik, S., Wolf, W., Wolfe, A., Li, S. Y. T., Yen, T. Y. "Performance Analysis of Embedded Systems." In: De Micheli, G., Sami, M. (eds) *Hardware/Software Co-Design*. NATO ASI Series, vol 310. Springer, Dordrecht (1996). https://doi.org/10.1007/978-94-009-0187-2_3

[3] Schoenit, J. K., "Optimizing Time Series Database Operations on Resource-Constrained Embedded Devices." (2024). https://open.library.ubc.ca/collections/undergraduateresearch/52966/items/1.0442407

[4] ObjectBox, "An Edge for Automotive." https://objectbox.io/connected-car-data-storage-and-sync/

[5] Schneider Electric, "Have You Heard of Industrial Edge Computing?" YouTube. https://www.youtube.com/watch?v=dSAxAahUWkE

[6] McObject, LLC, "McObject and Siemens Embedded Announce Immediate Availability of eXtremeDB/rt for Nucleus RTOS." Global Newswire. globenewswire.com/news-release/2022/02/23/...

[7] Montemagno, J., Stevens, A., "Saving Data Locally with SQLite for Meadow IoT Devices." Microsoft Learn. https://learn.microsoft.com/en-us/shows/on-dotnet/saving-data-locally-with-sqlite-for-meadow-iot-devices

[8] Ochando, F. J., Cantero, A., Guerrero, J. I., León, C., "Data Acquisition for Condition Monitoring in Tactical Vehicles: On-Board Computer Development." *Sensors (Basel)*, **23**(12):5645, 2023. doi: 10.3390/s23125645. PMID: 37420811; PMCID: PMC10301766.

[9] Anderson, T., "The Architecture of the Web: Performance and Scalability." *ACM Transactions on the Web*, **1**, 3 (1999). https://doi.org/10.1145/332833.332837

[10] Pinheiro, A. M. G., Abreu, J. F., Faria, R. P., "Optimizing Time Series Database Operations on Resource-Constrained Embedded Systems." Undergraduate thesis, University of British Columbia. https://open.library.ubc.ca/soa/cIRcle/collections/undergraduateresearch/52966/items/1.0442407

[11] IoT Analytics, "Number of Connected IoT Devices Worldwide 2025." *IoT Analytics*, 2020. https://iot-analytics.com/number-connected-iot-devices/

[12] UNREPO, "Characteristics and Challenges of Embedded Systems." *UNREPO*, 2023. https://www.unrepo.com/embedded/characteristics-and-challenges-of-embedded-systems

[13] Azul, "Download Azul JDKs." https://www.azul.com/downloads/

[14] Liu, Hao. "ARM-Based Embedded System Platform and Its Portability Research." Journal of Computer and Communications, vol. 11, no. 11, Nov. 2023 https://www.scirp.org/journal/paperinformation?paperid=129148